Morse:A functional query language and its semantic data model

M. Bouzeghoub

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 270

MORSE: A FUNCTIONAL QUERY LANGUAGE AND ITS SEMANTIC DATA MODEL

Mokrane BOUZEGHOUB

Février 1984

MORSE: A FUNCTIONAL QUERY LANGUAGE AND ITS SEMANTIC DATA MODEL

MOKRANE BOUZEGHOUB

SABRE Project, INRIA and University of PARIS VI

Domaine de Voluceau Rocquencourt B.P. 105

78153 Le Chesnay Cedex FRANCE

ABSTRACT: A Semantic Network Model is a triple SN(NC,AC,IC) characterized by the category of nodes CN, category of arcs AC and category of constraints IC. All these categories are carefully described with some basic predicates. To manipulate this Semantic Network, a Functional Language is proposed. It consists of a set of built-in functions, primitives and derived forms, and a set of functional forms which are expressions composed with the formers. Properties from both functional and derived forms are deduced and query examples illustrate the use of MORSE.

KEY-WORDS: Data Models, Semantic Networks, Relational Model, Functional Programming, Set Theory.

RESUME : Un réseau sémantique peut être défini comme un triplet SN(NC,AC,IC) caracterisé par la catégorie de noeuds CN, la categorie d'arcs AC et la catégorie de contraintes IC. Toutes ces catégories sont précisément decrites avec des predicats de base. Pour manipuler ce réseau, un langage de type fonctionnel est proposé. Il consiste en un ensemble de fonctions prédéfinies, les primitives et les formes dérivées, et un ensemble de formes fonctionnelles qui sont des expressions composées à l'aide des précédentes. Des propriétés sont déduites aussi bien sur les formes fonctionnelles que sur les formes dérivées et des exemples de requêtes illustrent l'utilisation de MORSE.

MOTS-CLES: Modèles de Données, Réseaux Sémantiques, Modèle Relationnel, Programmation Fonctionnelle,Théorie des Ensembles.

1- INTRODUCTION

Current research on Semantic Networks incorporates two trends in computer Science: Databases and Artificial Intelligence. Both domains are concerned with formally representing the knowledge we have of a given Real World. But unlike other structuring data models which are based on either a theory (relational model) or on standards (DBTG-CODASYL model), Semantic Networks suffer from a lack of this unity. A look at the literature in the field leads us to conclude that there are as many Semantic Networks as authors. This is due to the non-uniformity of the vocabulary; each author using the specialized language of the domain of application which interests him. If we take a closer look at the different models, however, we can extract a set of a very useful concepts adopted by most of them. These concepts will constitute the main framework of our semantic network. This paper aims to provide a powerful language to precisely describe and to manipulate this semantic network. In the second section, we define this semantic network as precisely and formally as possible. The third section reviews the essential elements of functional programming as defined by BACKUS [BACK78] as well as the derived applications for databases. The fourth and fifth sections specify our particular application, MORSE, of functional programming in order to manipulate the semantic network defined in the second section. The sixth section summarizes some general properties and constraints of the semantic network, which are described by our functional language.

2- DEFINITION OF A SEMANTIC NETWORK.

We call a network a set of labelled nodes and labelled arcs linking these nodes. The semantics of this network is expressed by a precise classification and identification of the nodes and arcs, and, possibly, by some constraints on each node or arc. A semantic network model may be defined by the triple

SN(NC,AC,IC) where NC represents the node categories, AC the arc categories and IC the integrity constraint categories.

2-1- NODE CATEGORIES

Each node in the SN belongs to one of the following categories:

* ATTRIBUTE * VALUE

* ENTITY * INSTANCE

The difference between the (ATTRIBUTE,VALUE) couple on the one hand and the (ENTITY,INSTANCE) couple on the other hand, lies in the atomicity of the concepts. We consider then, that attributes and values are atomic concepts, whereas entities and instances are molecular concepts built from the former. For example, if we are not concerned with the structure of the strings of characters, NAME and WATSON are atomic concepts (attribute and value respectively), while PERSON is a constructed concept built from NAME, AGE and ADDRESS. But a DATE may be considered as an entity if we are interested in its structure of DAY, MONTH and YEAR , and as an attribute if we consider it as a unique value. In this paper, we will not consider the case in which an object may be looked at as an entity (or instance) and as an attribute (or value) at the same time.

The difference between the (ATTRIBUTE,ENTITY) couple on the one hand and the (VALUE,INSTANCE) couple on the other lies in the descriptive role of the former and the referencial role of the latter. For example, any VEHICLE may be described by its MAKE, its LICENSE_NUMBER and its COLOR. But a particular vehicle may be referenced by values assigned to its MAKE, its LICENSE_NUMBER and its COLOR.

Each node will then always be defined according to these two orthogonal aspects. The first may be considered as an intentional definition and the second as an extensional definition. In the terminology of the Relational

Model, we find exactly the same concepts of attribute and value. An entity corresponds to CODD's relation and the instance of an entity corresponds to a tuple of a relation. But while a relational schema contains only the concepts of entity, attribute and functional dependencies between attributes, semantic networks enable us to have all of these concepts available at the same time in a schema as well as a great variety of arcs. The following sub-section deals with these different arcs.

2-2- ARC CATEGORIES

Arcs are placed into categories depending on whether they link nodes of the same or of different categories. We distinguish three categories of arcs which are portrayed in Figure 1. Each type of arc is defined by its category (or the node categories it links), its orientation and a predicate specifying its semantics. To identify each type of arc thus defined, we will give it a name which may be considered as an abstract of both its category and its predicate. Thus each type of arc will be defined as a Boolean function:

ARC-NAME: NODE-CATEG.1 X NODE-CATEG.2 -----> {TRUE,FALSE}

ARC-NAME(ni,nj) Γ = if the semantics given by the name of the arc exists between ni and nj and False otherwise. The first member of the couple (ni,nj) indicates the origin of the arc and the second member its target. With each arc, we may associate its inverse which we will only refer to if it is of interest in the following sections. Hereafter, we will review some of these different types of arcs. But first, let EN be the set of all entities, AT the set of all attributes, VA the set of all values of all attributes and IE the set of all instances of all entities.

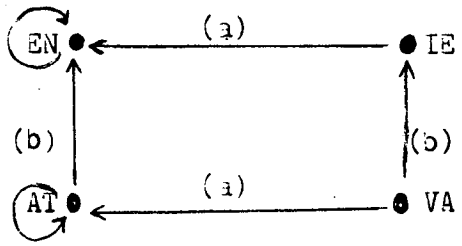


Fig. 1: Arcs categories.

a) ARCS LINKING ENTITIES/INSTANCES AND ATTRIBUTES/VALUES

Arcs linking entities and attributes describe the structure of an entity node by giving its components or specifying that a node is a part of the structure of another node. These are exactly the same relationships as those known in the factory under the name of "nomenclature relationships" expressed between products. In our case, they are represented by the two following arcs:

a : AT X EN -----> {T,F}
a(at,en) = T if at is a part of the structure of en.

ρ : EN X AT -----> {T,F}
ρ(en,at) = T if en is composed of at.

We say that these arcs define intentionally an entity. From these two kinds of arcs, we may redefine the concepts of aggregation given by [SMSM77] as the union of arcs "a" converging on the same node. A particularization is defined as the union of arcs "ρ" going out from the same node. In the following sections, we will call these arcs aggregation and particularization arcs or simply arcs a and ρ.

Note 1: The same arcs defined between EN and AT may be declared between IE and VA. Since each node belongs to a specific category, there is no confusion to declare these arcs between entities and attributes on one hand and between instances and values on the other hand.

Note 2: As with all nomenclature relationships, when an object x is built from another object y, we give what is called the gathering factor of the latter. This supplementary constraint is discussed in section [2-3].

b) ARCS LINKING ENTITIES/ATTRIBUTES AND INSTANCES/VALUES

These arcs associate a value with each given attribute node and an instance with each given entity node. We call them classification and instantiation arcs, respectively. They define an entity or an attribute by extension. An instantiation arc "i" is defined as:

i : EN X IE -----> {T,F}
i(en,ie) = T if ie is a possible occurrence of en.

Its inverse is the arc c which specifies that en is a possible abstraction of ie. In the same way, we define the following arcs between attributes and their values:

i : AT X VA -----> {T,F}
c : VA X AT -----> {T,F}.

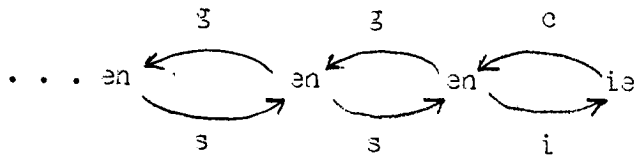
c) ARCS LINKING NODES OF THE SAME CATEGORY

Here, we will only consider the node categories EN and AT and subsequently look at the others.

(1) Generalization/Specialization arcs.

As for aggregation/particularization, generalization may be defined as the union of arcs g converging on the same node. Likewise, the specialization is defined as the union of arcs s going out from the same node. The g/s arcs may be defined according to two different but complementary view points:

- Relating to the classification/instantiation arcs: The generalization arc corresponds to a recursive application over EN of the classification arc. It is the same for the specialization arc in relation to the instantiation arc.



Using set theory terminology, the arc c expresses the membership relationship

($\in$) while the arc g expresses the inclusion relationship ($\subset$).

- Relating to the nodes : Generalization/Specialization arcs express the direction in which the inheritance rules are applied. Thus, if eni is a generalization of enj (or inversely, if enj is a specialization of eni) then enj inherits all the properties of eni. Formally, in this case, g/s arcs are defined as:

g : EN X EN -----> {T,F}
g(enj,eni) = T if eni gives its properties to enj.

s : EN X EN -----> {T,F}
s(eni,enj) = T if enj inherits the properties of eni.

These definitions of g/s arcs may be recursively applied over EN.

Note 1: Often, especially in Artificial Intelligence, g arcs are called "is-a" arcs.

Note 2: Sometimes, g/s arcs may likewise be defined as relationships linking a concept to a meta-concept.

Note3: These concepts of generalization/specialization are not significant, if declared, in the category of instances IE.

(2) Association arcs.

These are all arcs whose semantics is defined each time by any predicate other than those already defined. Generally, they are defined between two entities, two instances or an instance and an entity. In the example given in Figure 2, "enrolled", "loves" and "likes" correspond to this type of arc. Their semantics depends on each application to be described. Having no a priori knowledge of these arcs, we will hereafter represent them by the symbol r and call them application arcs or association arcs. For example, if an association is declared between two entities:

r : EN X EN -----> {T,F}
r(eni,enj) = T if there exists any association between eni and enj.

In fact, we will use r as a declarative constructor of associations, as we use the declarative statement TYPE to create new types in PASCAL language. For

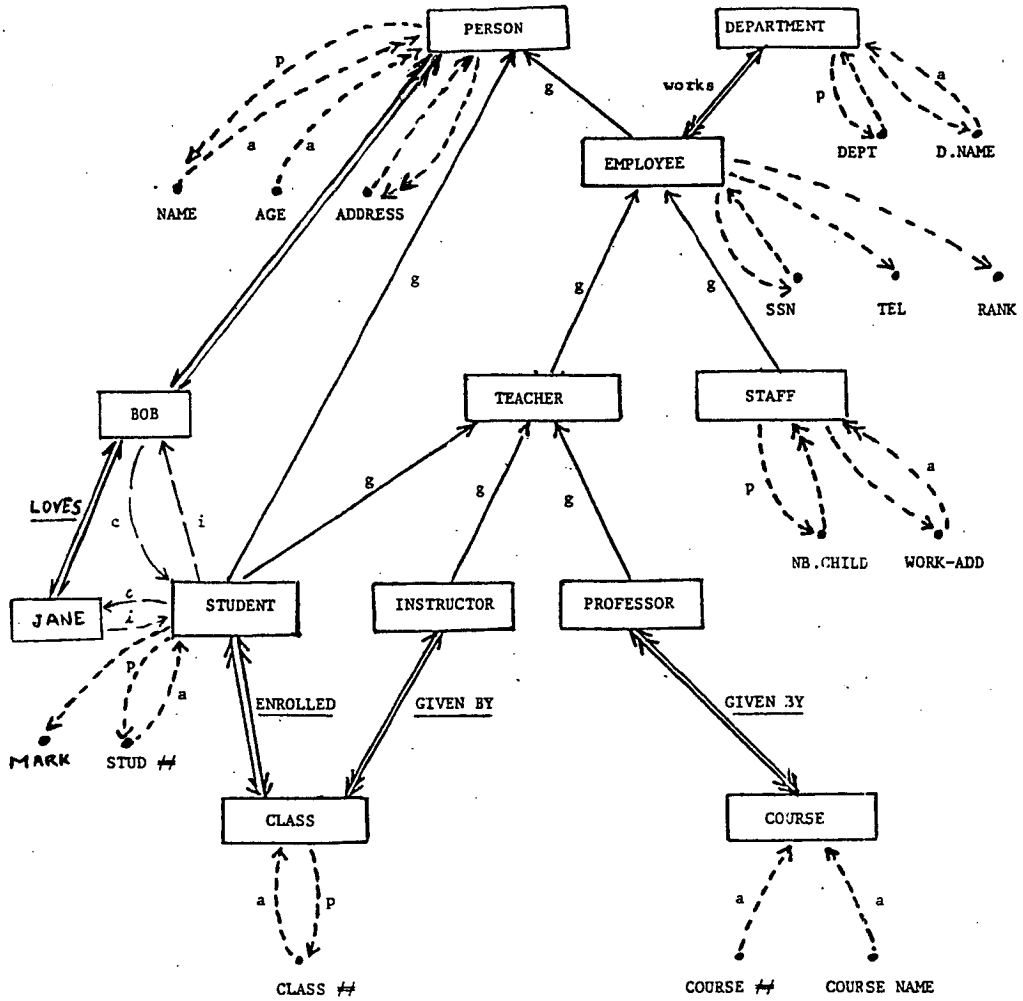


Fig. 2: An example of a semantic network.

example, to declare the application area "loves", "enrolled" and "likes", we use the following statements:

```
area-name(X,Y) => r(CATEG1,CATEG2)
```

where X and Y are node variables, r is a predefined predicate which states that area-name is an association and CATEG1 and CATEG2 are the categories for which X and Y are the instances. For example,

```
enrolled(X,Y) => r(EN,EN)
loves(X,Y) => r(IE,IE)
likes(X,Y) => r(IE,EN)
```

After these definitions, we may use these associations to describe facts of the real world exactly as it was done with basic area a/p, g/s and c/i:

```
enrolled(STUDENT,COURSE)
loves(BOB,JANE)
likes(BOB,PERSON)
```

Since each arc may have its inverse, we can declare it with the following definition: loves => inv(is-loved-by) where inv is a predefined predicate.

2-3- CONSTRAINT CATEGORIES.

A constraint is any predicate which can contribute to a more precise definition of a node or an arc. We will not examine here all the types of constraints but only those which seem to be a necessary complement to the network we have adopted. These constraints may be expressed either by additional nodes, additional arcs or any predicate specified by the MORSE language. We outline three types of constraints here:

a) THE INTERSECTION CONSTRAINT

In the example represented Figure 2, we show that EMPLOYEE may either be a TEACHER or a STAFF_MEMBER. We do not, however, specify whether these statements are exclusive or not. To capture this semantics, it would then be desirable to add a new constraint linking the s arcs going out from the node EMPLOYEE. This constraint may be represented by an arc which links two arcs

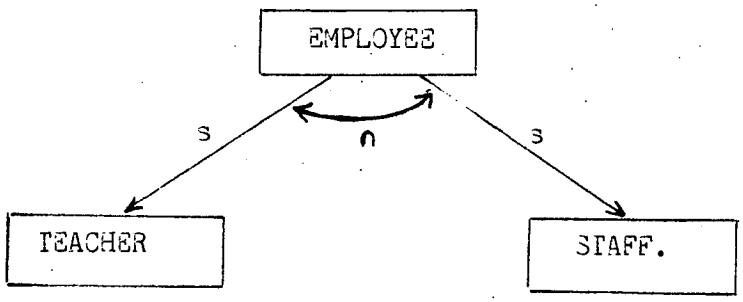


Fig. 3: An intersection arc.

as shown in Figure 3. We will see later that this constraint may be formally expressed by cardinalities.

We may adopt the same reasoning for STUDENT and INSTRUCTOR in relation to PERSON. But this constraint is too wide as it would allow us to say that a

STUDENT may be a PROFESSOR, which, naturally, is not exact in the university context. We must however absolutely specify that a STUDENT may be an INSTRUCTOR and vice versa. To express this constraint like the preceeding one, we must create a transitive closure of s arcs between INSTRUCTOR and

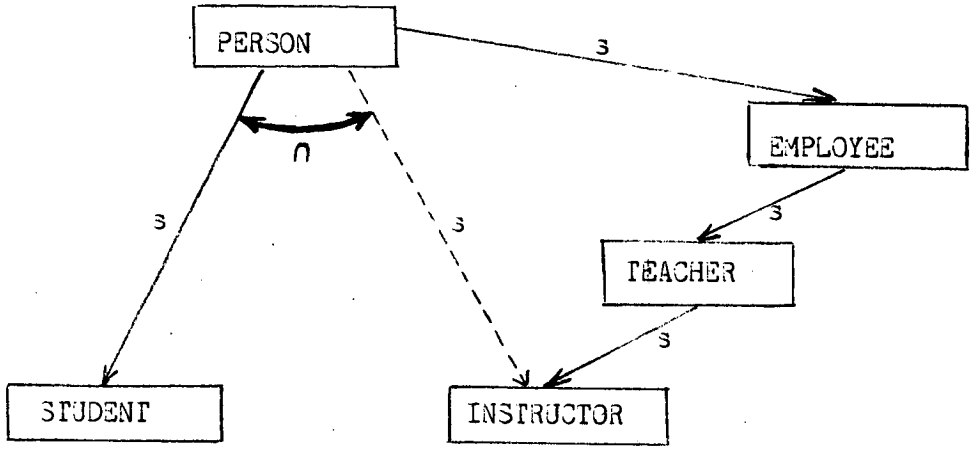


Fig. 4: A transitive closure of s.

PERSON (see Figure 4). This solution creates a redundant arc which we do not know whether this arc is desirable or not at the current stage of our investigation.

The two preceeding solutions are only possible when a generic node exists. Thus, in the case in which a PERSON was not created in Figure 2, this way of representing such a constraint fails. To palliate this problem, we have tested three solutions:

(1) We may create a virtual generic node every time this type of constraint arises between two nodes and find ourselves back at the first case (Figure 5).

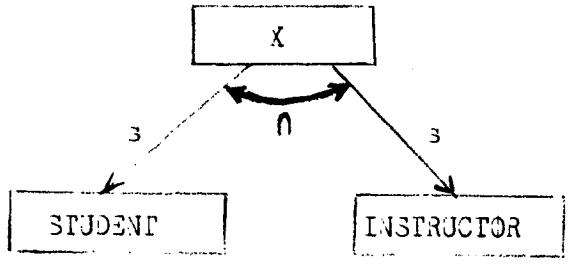


Fig. 5: A virtual generic node.

But this solution does not work because it is not easy to specify the precise semantics of X. If X stands for PERSON, the constraint is plausible but if X stands for TENNIS_CLUB_MEMBERS, for example, we may suppose that a member is

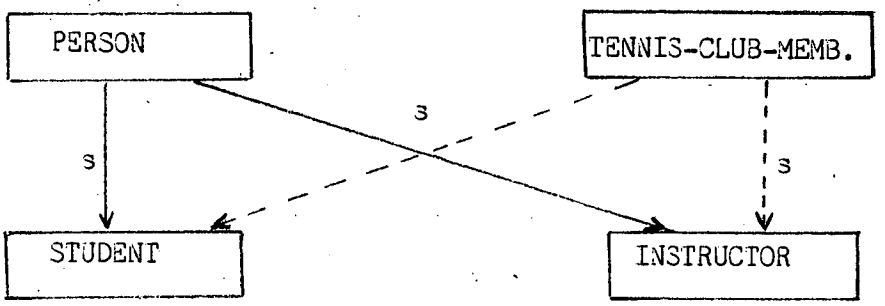


Fig. 7: A conflictual constraint.

exclusively enrolled as a STUDENT or as an INSTRUCTOR; then the constraint is wrong. If the two contexts may exist together, it really becomes an ambiguous situation (Figure 6).

(2) We consider STUDENT and INSTRUCTOR themselves as generic nodes and create an intersection node STUD_INST which is both the specific node of STUDENT and

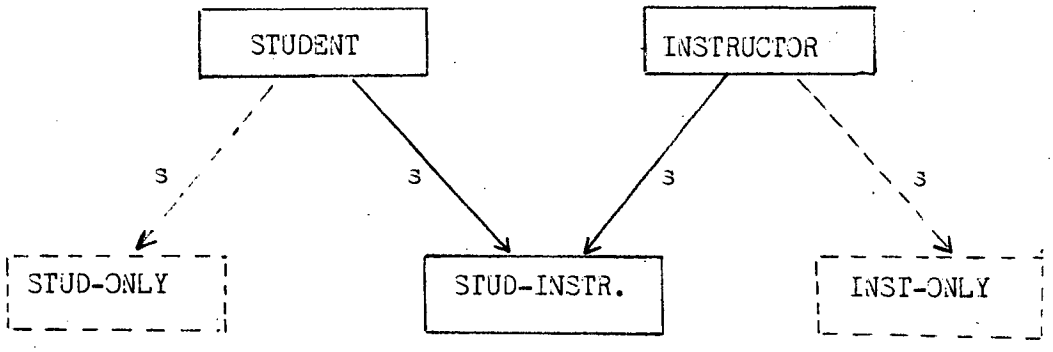


Fig. 7: An intersection node.

INSTRUCTOR (Figure 7). This corresponds to the notion of convergent relation of [LEGE78]. But to be complete, we must create the complementary specific nodes STUD_ONLY and INST_ONLY. The same remarks about the redundance of arcs holds here for the redundance of nodes.

(3) Third, we may create a new type of arc called an intersection arc which links STUDENT and INSTRUCTOR:

STUDENT<-mb-->INSTRUCTOR

The semantics of this arc is : a STUDENT may be an INSTRUCTOR and an INSTRUCTOR may be a STUDENT. This arc, labelled mb, is defined by the following predicate:

mb(STUDENT,INSTRUCTOR) = T if $\exists x \in$ IE / i(STUDENT,x) = T
and i(INSTRUCTOR,x) = T.

This arc could easily be mapped to the previous representation. This is the solution that we will keep hereafter and that we will generalize even to the preceeding cases.

b) THE FUNCTIONAL DEPENDENCY CONSTRAINT

Defined between two attributes, this constraint expresses the same semantics as the functional dependency of the Relational Model, even if the context of relation in which it has been defined in the Relational Model is not significant here. In fact, the semantics we would like to express here is the following: given two objects x and y, we say that y is functionally dependent on x if, from the knowledge of x, we can deduce that of y. We will represent these arcs as follows:

d : AT X AT -----> {T,F}
d(ati,atj) = T if atj is functionally dependent on ati
(or if ati determines atj).

The same predicates may be declared between two values, two entities or two instances.

c) THE CARDINALITY CONSTRAINT

this constraint specifies the number of arcs of the same type outgoing the same node or, in other words, how many nodes are connected to a given node

through the same type of arc. If this number varies between two bounds, then this constraint will be represented by a couple [m,n].

To illustrate this constraint, let us take a classic example from management of nomenclature relationships of products. When a product P1 enters into the composition of another product P2, we must specify the gathering factor, the number of copies of P1 necessary to produce P2. In our case, if an entity PERSON is built from the attributes NAME, FIRST-NAME and ADDRESS, we must specify that a given person may have one name, one first name and three addresses for example. This is shown in Figure 2 by simple arrow or double arrows on the p arcs. But this number may vary from one person to another. Thus we will represent this by a pair of values [m,n] specifying the minimum and the maximum gathering factors. We will call this pair of values cardinalities. We may generalize these cardinalities to each type of arc but they do not have the same meaning nor the same importance for all of them. A more formal definition of this constraint will be given in section 6.

3- REVIEW OF FUNCTIONAL PROGRAMMING AND LANGUAGES

According to Backus, a Functional Language (FL) must contain the following components:

(1) A set O of objects which are either atoms or sequences of objects. The undefined object, $\underline{\perp}$, and the empty sequence, $\underline{\phi}$, are included in this set as well as the logical values True (T) and False (F).

(2) A set F of primitive functions like +,-,x,and,or,not,...

(3) A set FF of functional forms which allow to define new functions by composition and construction laws applied to primitives. For example, $f.g:x \equiv f(g(x))$ is a composition while $(f,g):x \equiv \langle f(x),g(x)\rangle$ and $(p \rightarrow f;g):x \equiv$ if $p(x)$ then $f(x)$ else $g(x)$ are constructions. Backus gives more than twenty composition and construction laws [BACK78].

(4) A set D of definitions that define some new functions using functional forms and assign a name to each. A definition has the form: Def l = r where l is the new function and r a functional form (composition or construction).

Inspired by this functional programming (FP) and the other applicative languages such as LISP and APL, several researchers in DB area have proposed functional data models and functional query languages. We shall refer hereafter to Buneman s and Shipman's works on FQL and DAPLEX, two functional query languages. The related data models suggest that most relationships among data will be represented as mathematical functions defined over basic or constructed domains. Thus the objects PERSON and NAME, rather than being an entity and an attribute of this entity respectively, will be declared as functions. In DAPLEX [SHIP81], these declaration statements are:

```
DECLARE PERSON() ===>> ENTITY
DECLARE NAME(PERSON) ===> STRING
```

ENTITY and STRING are basic domains, PERSON and NAME are functions which, in turn, could be built-in domains over which other functions could be declared. For example:

```
DECLARE STUDENT() ===> PERSON
```

PERSON and STUDENT are zero-adic functions while NAME is a mono-adic function. PERSON represents a set of entities (double arrow), NAME represents a string of characters.

In FQL, the previous statements are written as follows:

```
PERSON:          ---> !ENTITY
NAME:   PERSON   ---> STRING
STUDENT:         ---> PERSON
```

!ENTITY denotes a set of entities. The zero-adic functions have no known origin domain. An FQL query is a functional form which may have the form:

```
!PERSON.*NAME;
```

where *NAME denotes the set of names of people. The periode stands for a composition of functions. The next query

```
!PERSON.|MARRIED.*NAME;
```

gives the set of names of married people. We have supposed that MARRIED was previously defined as a Boolean function:

```
MARRIED: PERSON ---> BOOL
```

In DAPLEX, the first query will be written as follows:

```
For each PERSON print NAME(PERSON);
```

and the second:

```
For each PERSON
    such that MARRIED(PERSON)=True
    print NAME(PERSON);
```

The two statements above show that DAPLEX has embedded the functional forms in a procedural language whereas FQL statements are only declarative. These two applications of FP to DB lead their authors to redefine a Database as a collection of primitive functions and functional forms [BUNF82]. These functions depend on each application and are defined by the end_user.

In the following sections, we shall propose a language of the same class and characterized by two aspects. First, it has a set of built-in functions and a set of variable application functions every time defined by the end-user. Second, it operates on a Semantic Network which represents a set of facts describing the Real World.

4- SUMMARY OF THE MORSE LANGUAGE

We can define MORSE language as a tuple (SN,PF,FF,DF) where SN is the set of objects which correspond to that of the semantic network defined previously, PF is the set of primitive functions, FF is the set of functional forms and DF is the set of derived forms.

4-1- THE SET OF OBJECTS

This set is composed of all categories of nodes of the Semantic Network

defined in section 2 including the empty set, $\underline{\phi}$, the undefined object, $\underline{!}$, and the logical values T and F.

4-2- THE PRIMITIVE FUNCTIONS

a) CONSULTING PRIMITIVES

With each type of arc f, defined from a given domain $\mathcal{D}$ to a given domain $\mathcal{A}$, we define a primitive function F over the domains $\mathcal{D}$ and $\mathcal{P}(A)=2^{\mathcal{A}}$ and which scans the arcs f. $\mathcal{P}(A)$ is the set of parts of A.

$F: \mathcal{D} \text{ -----> } \mathcal{P}(A)$
$F(x) = A' = \mathcal{P}(A) \;/\; \forall y \in A',\ f(x,y) = T$
$F(x) = \underline{\phi}$ if there does not exist any y connected to x by f.
$F(x) = \underline{!}$ if x does not belong to $\mathcal{D}$.

For example, from the arc p, we define the function P as follows:

$P: EN \text{ -----> } \mathcal{P}(AT)$
$P(en) = AT' = \mathcal{P}(AT) \;/\; \forall at \in AT',\ p(en,at) = T$

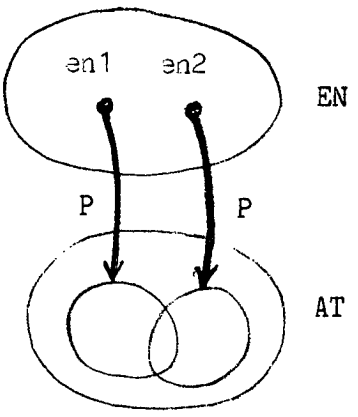


Fig. 8: The primitive function P.

This function returns the set of attributes "at" linked to an entity "en" by the p arcs. In Figure 8, the labelled hiper-arc P represents a cluster of p arcs. In the same way, we may define the function I:

$I: EN \text{ -----> } \mathcal{P}(IE)$
$I(en) = IE' = \mathcal{P}(IE) \;/\; \forall ie \in IE',\ i(en,ie) = T$

which returns the all instances "ie" of the entity "en" (Figure 9). As with P, I illustrates a cluster of i arcs in Figure 9. Each primitive F may have either a simple node or a set of nodes as arguments:

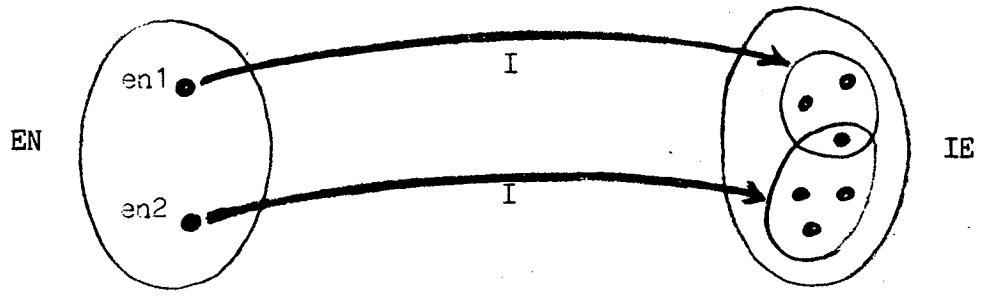


Fig. 9: The primitive function I.

$$F(\{X1,X2,\ldots,Xn\}) = \{F(X1),F(X2),\ldots,F(Xn)\}$$

provided that F be defined for each Xi. For example,

$$P(\{PERSON,EMPLOYEE\}) = \{P(PERSON),P(EMPLOYEE)\}$$
$$= \{\{NAME,FIRST\_NAME,ADDRESS\},\{SSN,TEL,RANK\}\}$$

This equation seems like a combination of the primitive F with the famous MAPCAR concept of LISP. The following list summarizes the primitives corresponding to the basic ares:

A: AT -----> P(EN)	or	A: VA -----> P(IE)
G: EN -----> P(EN)		
C: IE -----> P(EN)	or	C: VA -----> P(AT)
R: EN -----> P(EN)	or	R: IE -----> P(IE)
	or	R: IE -----> P(EN)

In the same way, we can declare the symmetric functions P, S, I and R-1.

o) CONSULTING WITH RESTRICTIONS

Each set, resulted from the previous primitives, may be filtered by the following predicates:

$$LT(F(X);Y) = \{x \in F(X) \ / \ x<Y\}$$

where LT stands for "less than" and Y the value with which the members of F(X) are compared. In the same way, we may define the functions GT (>), EQ (=), GE (>=), LE (<=) and NE (≠). For example, to select all the ages less or equal 20, we write: LE(I(AGE);20). To simplify the functional forms, we introduce the usual combination of Y1<=x<=Y2 as a function and we will denote it BT (for between).

$$BT(F(X);(Y1,Y2)) = \{x \in F(X) \ / \ Y1<=x<=Y2\}$$

For example, to select all the marks less than 10, we write:

$$LE(I(MARK);10)$$

and to select the all salaries between 5000 and 7000, we write:

$$BT(I(SALARY);(5000,7000)).$$

Note: If succeeded, the function EQ(F(X);Y) returns Y as a result. It seems like a redundant function of f(X,Y). But if the second one returns a logical value as a result, the first one checks the logical predicate and returns the values Y or $\underline{\phi}$ whether the predicate is true or not. Depending on the situations, it will be more interesting to use the first one or the second one.

c) UPDATING PRIMITIVES

An updating of a semantic network consists in the creation or the deletion of some arcs, preceeded or followed by the creation or deletion of the corresponding nodes possibly. Nodes are created by the statement i(CATEGORY,x) where CATEGORY stands for one of the fourth categories ENTITY, ATTRIBUTE, INSTANCE and VALUE. Arcs are created by the statement f(x,y) where f is the predicate which defines the arcs linking x and y. If the nodes x and y already exist, then only the predicate f is created between them; otherwise both the nodes and the predicate are created. The deletion of such an arc is accomplished by the statement NOT(f(x,y)). The deletion of a node assumes that all its related arcs are already deleted and is done by the statement NOT(i(CATEGORY,x)). According to the definition of f, if x and y are mutually incompatible, f takes the undefined value $\underline{!}$.

4-3- FUNCTIONAL FORMS

They essentially coincide with the set operations union, difference and intersection plus an original composition called product applied to the primitives. The commutative, associative and distributive properties of these

operations characterize the composition laws and have the same roles as the syntactic and the semantic rules in the usual programming languages. They also permit to replace one query expression by another which could be easier to evaluate. A table summarizing the allowable compositions is given in the appendix. We will likewise consider several constructions with Boolean functions as functional forms even if they are more writing conventions than compositions.

a) UNION OR SUM OF FUNCTIONS

If Y1 and Y2 denote the sets resulting from the functions F1 and F2, respectively, and if they belong to the same category of nodes, then, by definition:

I-1 $(F1+F2)(X) = F1(X) + F2(X) = Y1 \cup Y2$
$= \{y/\ f1(X,y)=T \text{ or } f2(X,y)=T\}$

where U represents the union of sets. If Y1 and Y2 do not belong to the same category of nodes, we will say that the sum is undefined and we will represent it by the symbol $\underline{|}$. For example,

$(G+S)(EMPLOYEE) = G(EMPLOYEE) + S(EMPLOYEE)$
$= \{PERSON\} \cup \{TEACHER, STAFF\_MEMBERS\}$
$= \{PERSON, TEACHER, STAFF\_MEMBERS\}.$

$(G+P)(EMPLOYEE) = G(EMPLOYEE) + P(EMPLOYEE) = \underline{|}$

because the results of G and P are incompatible (do not belong to the same category of nodes).

PROPERTIES OF THE SUM

- The sum is commutative: since the order does not matter in sets, we may write:

I-2 $(F1+F2)(X) = (F2+F1)(X) = F1(X) + F2(X)$

- The sum is associative:

I-3 $((F1+F2)+F3)(X) = (F1+(F2+F3))(X) = F1(X) + F2(X) + F3(X)$

- The sum has a neutral element: to express everything in a functional form, we will assume that the objects Xi are zero-adic functions whose results are Xi themselves. Among these functions, there exist two particular, $\underline{\phi}$ and $\underline{|}$, which produce the empty set and the undefined object respectively. The function $\underline{\phi}$ is the neutral element of the sum:

I-4 $F(X) + \underline{\phi} = F(X)$

- The sum does not produce symetric elements but it has an absorbing element, $\underline{|}$:

I-5 $F(X) + \underline{|} = \underline{|}$

from which we may deduce:

I-6 $\underline{\phi} + \underline{|} = \underline{|} + \underline{\phi} = \underline{|}$

I-7 $\underline{|} + \underline{|} = \underline{|}$ and

I-9 $\underline{\phi} + \underline{\phi} = \underline{\phi}$

b) DIFFERENCE OF FUNCTIONS

We define the difference of funtions as follows:

II-1 $(F1-F2)(X) = F1(X) - F2(X)$
$= \{y/\ f1(X,y)=T \text{ and } f2(X,y)=F\}.$

For example,

S(PERSON) - S(TEACHER) = {STUDENT,EMPLOYEE} - {STUDENT,PROF} = {EMPLOYEE}

PROPERTIES OF THE DIFFERENCE

- The difference is not commutative. If Y1 and Y2 are the resulting sets of

F1 and F2 respectivelly,

$$\left.\begin{array}{l}(F1\text{-}F2)(X) = \{y/\ y \in Y1 \text{ and } y \in Y2\} \\ (F2\text{-}F1)(X) = \{y/\ y \in Y2 \text{ and } y \in Y1\}\end{array}\right\} \Rightarrow/\Leftarrow \text{II-2}\quad (F1\text{-}F2)(X) = (F2\text{-}F1)(X)$$

But it is easy to demonstrate that they are associative:

II-3 $((F1\text{-}F2)\text{-}F3)(X) = (F1\text{-}(F2\text{-}F3))(X) = F1(X) - F2(X) - F3(X)$

- The difference has $\underline{\phi}$ as a right neutral element:

II-4 $F(X) - \underline{\phi} = F(X)$ but $\underline{\phi} - F(X) = \underline{\perp}$

- The difference has $\underline{\perp}$ as an absorbing element:

II-5 $F(X) - \underline{\perp} = \underline{\perp} - F(X) = \underline{\perp}$

c) INTERSECTION OF FUNCTIONS

It is the common sens of intersection of sets. We denote it:

III-1 $(F1xF2)(X) = F1(X) \times F2(X)$
$= \{y/\ f1(X,y)=T \text{ and } f2(X,y)=T\}$

As for the sum, we assume that the results of F1 and F2 are compatible, otherwise the intersection of F1 and F2 is undefined. This intersection is empty if the resulting sets of F1 and F2 are compatible but disjoint. For example,

$(PxS)(EMPLOYEE) = P(EMPLOYEE) \times S(EMPLOYEE) = \underline{\perp}$
$(GxS)(EMPLOYEE) = G(EMPLOYEE) \times S(EMPLOYEE) = \underline{\phi}$
$S(PERSON) \times G(TEACHER) = \{EMPLOYEE\}$

PROPERTIES OF THE INTERSECTION

- The intersection is commutative and associative but has not a neutral element. However, it has two absorbing elements $\underline{\phi}$ and $\underline{\perp}$.

III-2 $F(X) \times \underline{\phi} = \underline{\phi}$

III-3 $F(X) \times \underline{\perp} = \underline{\perp}$

If $F(X) = \underline{\perp}$, we assume that III-4 $\underline{\perp} \times \underline{\phi} = \underline{\perp}$

- The intersection is distributive with the sum: This is the well known result in set theory.

III-5 $((F1+F2)xF3)(X) = (F1(X) \times F3(X)) + (F2(X) \times F3(X))$

III-6 $(F1x(F2+F3))(X) = (F1(X) \times F2(X)) + (F1(X) \times F3(X))$

d) THE PRODUCT OF FUNCTIONS

If Y1 and Y2 are the resulting sets of F1 and F2 respectively, the product of F1 and F2 is defined as:

IV-1 $(F1*F2)(X) = F1(F2(X)) = F1(Y2) = Y1$
$= \{y1/\ ]y2,\ f1(X,y1)=T \text{ and } f2(y2,y1)=T\}$

provided that F1 be defined on Y2, otherwise the result is undefined and will be denoted $\underline{\perp}$. For example,

S(G(EMPLOYEE)) = S(PERSON) = {EMPLOYEE,STUDENT}

When the same function is recursively applied n times, it is denoted:

IV-2 $F^{(n)}(X) = F(F(F(...F(X))...))$

For example, $G^{(2)}(PROF)$ = {EMPLOYEE}. Generally, if we do not know the indice n, we will thus denote this function as: $F^*(X)$ and will then call it the <u>transitive target</u> of the arc f. For example, $G^*(PROF)$ = {PERSON} is the transitive target of g going out from PROF. This notation is a simplified form of the recursive notation proposed by Backus:

F1=(C-->F2;F1) <==> F1= if C is True
then F2 (exit condition)
else F1 (recursive call).

In our adaptation, the boundary condition, C, always satisfies to the predicate $F^*(X)=\underline{\phi}$ in the case of $F^*(X)$ or to the predicate k=n in the case of $F^{(n)}(X)$. The recursive call is represented by the symbol * or the indice n. The result of the function is the last set Y obtained before $\underline{\phi}$ or at the indice k=n. This functional form is especially applicable to those functions defined with transitive arcs like g, s and d or any transitive association

defined by the end-user. If we assume that

IV-3 $F^{(0)}(X) = X$ and IV-4 $F^{(1)}(X) = F(X)$

then a useful functional form could be derived from I-2 and IV-2:

$$\text{IV-5} \quad F^{+}(X) = \overset{*}{\bigcup_{k=0}} F^{k}(X)$$
$$= F^{(0)}(X) + F^{(1)}(X) + F^{(2)}(X) + \ldots\ldots + F^{*}(X)$$
$$= X + F(X) + F(F(X)) + \ldots + F(F(F\ldots F(X)\ldots))$$

This functional form searches a tree and returns the set of all its nodes whereas $F^{*}(X)$ gives only the set of its leaves. F^{+} is called the transitive closure of the arc f. For example, to show the usefulness of this function, consider $G^{+}(PROF)$. If we combine this functional form with the primitive P, then it yields the set of all attributes of PROF, both specific attributes and inherited attributes:

$P(G^{+}(PROF))$ = {OFFICE_NUM, SSN, TEL, RANK, NAME, AGE, ADDRESS}

PROPERTIES OF THE PRODUCT

- The product is not commutative. Indeed, if Y1 and Y2 are the results of F1 and F2 respectively, we will have:

$$\left.\begin{array}{l}(F1*F2)(X) = F1(F2(X)) = Y1\\ (F2*F1)(X) = F2(f1(X)) = Y2\end{array}\right\} \Longrightarrow \text{IV-6} \quad (F1*F2)(X) \neq (F2*F1)(X)$$

- The product of functions is associative:

IV-7 $(F1*(F2*F3))(X) = ((F1*F2)*F3)(X) = F1(F2(F3(X)))$.

- The product of functions does not have a neutral element and then does not produce symmetric elements. But, by definition, it has $\underline{\perp}$ as an absorbing element:

IV-8 $F(\underline{\perp}) = \underline{\perp}$

Also by definition, we will say that:

IV-9 $F(\underline{\phi}) = \underline{\perp}$

- The product is distributive with the sum: As we have previously said, if we consider the objects, Xi, as zero-adic functions and F(Xi) as a product of functions, then by definition, the product is distributive with the sum. Indeed, the previous definition of the sum

$$(F1+F2)(X) = F1(X) + F2(X)$$

may be perceived as a right distributivity of the product with the sum. Likewise,

$$F(X1+X2) = F(X1) + F(X2)$$

may be perceived as the left distributivity. Then,

IV-10 $((F1+F2)*F3)(X) = (F1+F2)(F3(X)) = F1(F3(X)) + F2(F3(X))$

and

IV-11 $(F1*(F2+F3))(X) = F1((F2+F3)(X)) = F1(F2(X)+F3(X)) = F1(F2(X)) + F1(F3(X))$

Applying the commutative property of the sum and the distributive property of the product with the sum, we can easily demonstrate the following results:

IV-12 $(F1+F2)(X1+X2) = F1(X1) + F2(X1) + F1(X2) + F2(X2)$

and

IV-13 $(F1*F2)(X1+X2) = F1(F2(X1)) + F1(F2(X2))$

and

IV-14 $(F1xF2)(X1+X2) = F1(X1+X2) \times F2(X1+X2) = (F1(X1)+F1(X2)) \times (F2(X1)+F2(X2))$

- The product is not distributive with the difference:

IV-15 $F1*(F2-F3))(X) = F1(F2(X) - F3(X)) \neq F1(F2(X)) - F1(F3(X))$

- The product is right distributive with the intersection:

IV-16 $((F1xF2)*F3)(X) = (F1xF2)(F3(X)) = F1(F3(X)) \times F2(F3(X))$

but it is not left distributive:

IV-17 $(F1*(F2xF3))(X) = F1((F2xF3)(X))$
$= F1(F2(X) \times F3(X))$
$\neq F1(F2(X)) \times F1(F3(X))$

e) CONSTRUCTION ON THE ARCS

If f is an arc and yi and zi are the elements of the sets Y and Z respectively, then the following statements are correct, by definition:

V-1 $f(Y,z1) \quad <===> \begin{cases} f(y1,z1) \\ f(y2,z1) \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \\ f(yn,z1) \end{cases}$

V-2 $f(y1,Z) \quad <===> \begin{cases} f(y1,z1) \\ f(y1,z2) \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \\ f(y1,zn) \end{cases}$

V-3 $f(Y,Z) \quad <===> \begin{cases} f(y1,z1) \\ f(y2,z1) \\ \cdots\cdots\cdots \\ f(yn,z1) \\ f(y1,z2) \\ \cdots\cdots\cdots \\ f(yn,zn) \end{cases}$

If Y and Z are the result sets of F1(X1) and F2(X2) respectively, then

V-4 $f(Y,Z)$ is $\begin{cases} \text{True} <===> \forall yi \in Y, \forall zi \in Z, f(yi,zi) = T \\ \text{False} <===> \exists yi \in Y \text{ or } \exists zi \in Z \ / \ f(yi,zi) = F \end{cases}$

We may also write:

V-5 $f(Y,Z) \quad <===> \quad f(F1(X1),F2(X2))$

Now, we give three examples of queries. The first one returns the young students (e.g. those whose the age is less than 20) and the second returns the good students (e.g. those whose the mark is greater than 14). The third query returns the students who are young but not good.

i I(EQ(A(AGE);STUDENT)) x A(LT(I(AGE);20))
ii I(EQ(A(MARK);STUDENT)) x A(GT(I(MARK);14))
iii (I(EQ(A(AGE);STUDENT)) x A(LT(I(AGE);20)))
- (I(EQ(A(MARK);STUDENT)) x A(GT(I(MARK);14))).

4-4- DERIVED FORMS

Functional forms are obtained by composition (sum,product,...) of primitive functions. Derived forms seem more like primitives than like functional forms since they are defined with the basic predicates (arcs). But instead of using one predicate to define one function as it was done for primitives, we use a combination of several predicates. We distinguish two categories of derived forms, the first order derived forms and the second order derived forms. We will summarize them hereafter.

a) THE FIRST ORDER DERIVED FORMS

From the previous primitives, we may define new sub-categories of objects over which we will define new functions. For example, I(X) and S(X) determine two sets of objects over which we can define the S' function as follows:

S': $I(X) \longrightarrow \mathcal{P}(S(X))$
$S'(x) = \{y/\ i(X,x)=T \text{ and } s(X,y)=T \text{ and } i(y,x)=T\}$

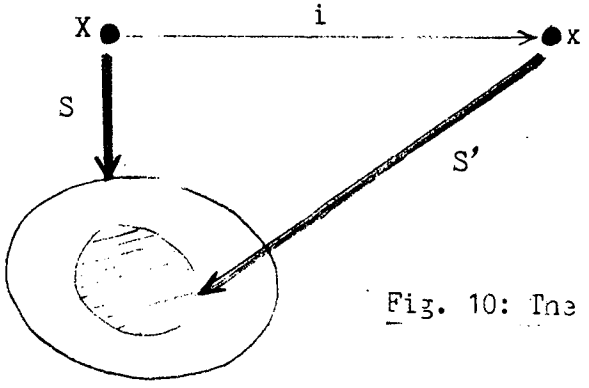


Fig. 10: The first order derived function S'

This definition could be represented by the Figure 10. We will call this function the derived form of S with respect to I and we will denote it $S'(x;I(X))$. For example, S'(WATSON;I(PERSON)) returns the actual specializations of the person WATSON, while S(PERSON) returns the all possible specializations of any PERSON. We may thus associate to each primitive its derived form, but this form is not always of interest. The following

equations generalize the definition of the first order derived forms.

$$F2' : F1(X) \longrightarrow \mathcal{P}(F2(X))$$
$$F2'(x;F1(X)) = \{y/\ f1(X,x)=T \text{ and } f2(X,y)=T \text{ and } f1(y,x)=T\}$$

The expression of the above statement may be simplified since we only study the derived forms with respect to I. Then,

$$F2'(x;I(x)) \equiv F2'(x;X)$$

With this simplification, the previous examples could be written as follows:

S'(WATSON;PERSON).

b) THE SECOND ORDER DERIVED FORMS

Also in this section, we assume that the derived forms are defined with respect to I. These forms are denoted F'' and are obtained by the following definition:

$$F''(x;(X,Y)) = \{y/\ f(X,Y)=T \text{ and } i(X,x)=T \text{ and } f(x,y)=T \text{ and } i(Y,y)=T\}$$

This definition is illustrated by the Figure 11.

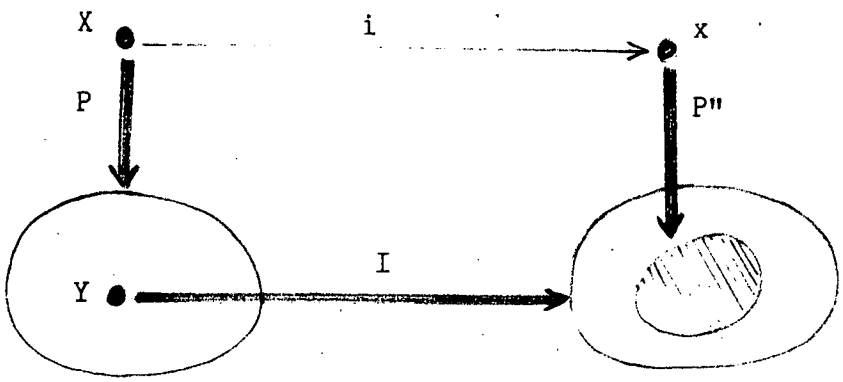


Fig. 11: The second order derived function P".

For example, P'' may be defined as follows:

$$P''(x;(X,Y)) = \{y/\ p(X,Y)=T \text{ and } i(X,x)=T \text{ and } p(x,y)=T \text{ and } i(Y,y)=T\}$$

If x stands for WATSON, X for PERSON and Y for ADDRESS then, P''(WATSON;(PERSON,ADDRESS)) returns the set of addresses of the person Watson.

c) COMPARING FUNCTIONAL FORMS AND DERIVED FORMS

The derived form S'(WATSON;PERSON) (Figure 12a) can easily be mapped to the following functional form (Figure 12b):

C(EQ(I(PERSON);WATSON)) x S(PERSON).

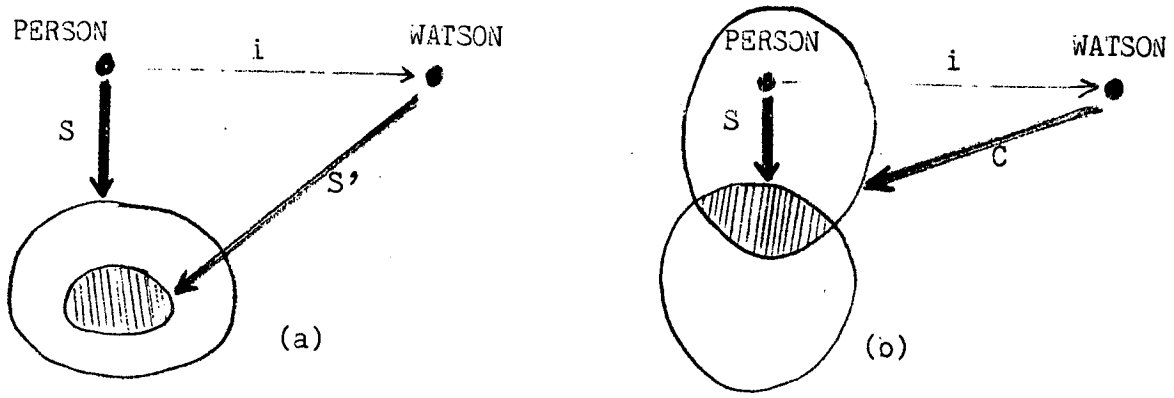


Fig. 12: First order derived form and functional form.

Notice the concision of the derived form compared to the functional form. The mapping of P''(WATSON;(PERSON,ADDRESS)) (Figure 13a) is (Figure 13b):

P(EQ(I(PERSON);WATSON)) x I(EQ(P(PERSON);ADDRESS))

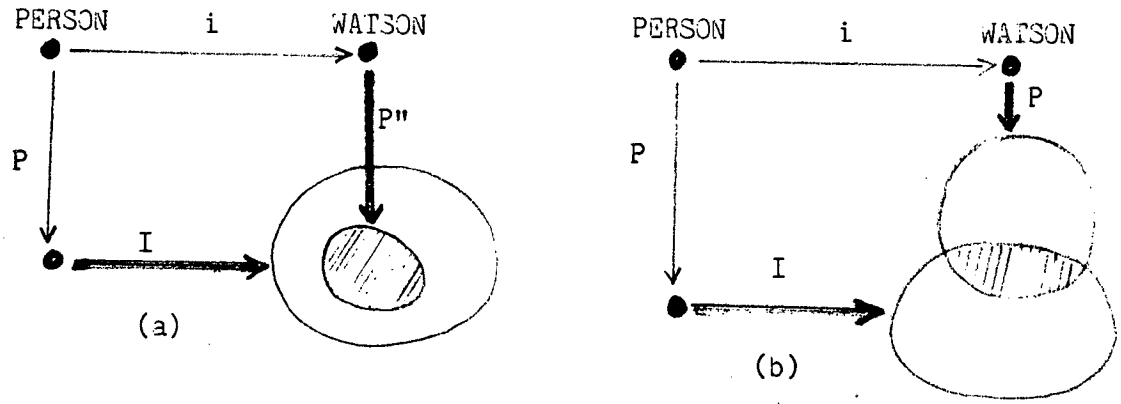


Fig. 13: Second order derived form and functional form.

Choosing derived forms instead of functional forms, the previous queries which return young students and good students can be expressed as follows:

YOUNG_STUD: A''(LT(I(AGE);20);(AGE,STUDENT))
GOOD_STUD: A''(GT(I(MARK);14);(MARK,STUDENT))

As with primitives, derived forms may be combined each other or with functional forms using the composition lows defined earlier.

5- THE EXTENSION OF THE LANGUAGE WITH NEW FUNCTIONS.

We have already stated that to declare new arcs (associations), we use the statements:

```
enrolled(X,Y) => r(EN,EN)
loves(X,Y) => r(IE,IE)
likes(X,Y) => r(IE,EN)
```

To define the corresponding primitives to these arcs, we use the following statements:

```
        primitive_name => R(arc_name)
or      primitive_name => R*(arc_name)
```

R^* is used for transitive arcs. For example, if we use lower case for arcs and upper case for primitves,

```
ENROLLED(X) => R(enrolled)
LOVES(X) => R*(loves)
LIKES(X) => R(likes)
```

With each functional form (composition of primitives or derived forms), we may associate a name and subsequently refer to this functional form by its name. This is especially useful when the functional form is a complex equation which is referred to too frequently. It is also a good way to introduce variables in functional forms. Notice that a definition establishes an equivalence between a symbol and an equation only but does not imply the evaluation of this equation. We will use one of the following statements to declare these names:

(1) name => ff where ff is the functional form. For example, STUD_ATT=P(G^*(STUDENT)). Then we may write I(STUD_ATT) instead of I(P(G^*(STUDENT))). The set of students whose age is less than 20 is called YOUNG_STUDENTS and is defined as follows (Figure 14):

```
YOUNG_STUDENTS => I(EQ(A(AGE);STUDENT)) x A(LT(I(AGE);20))
```

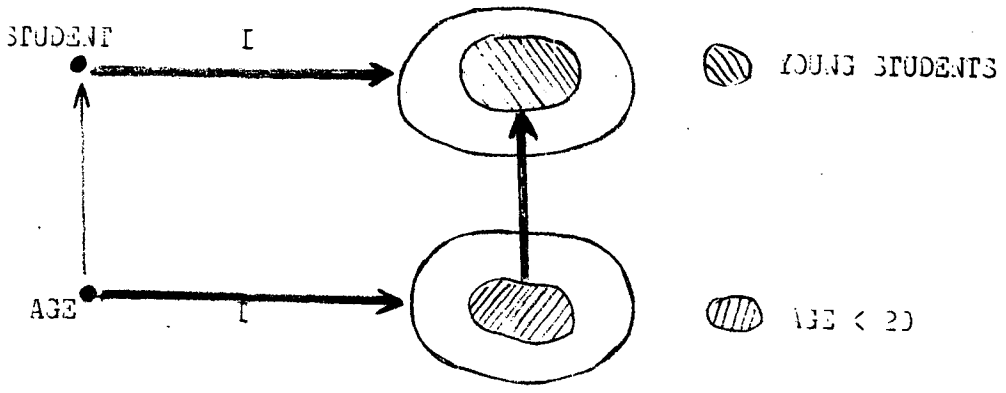


Fig. 14: Selection of young students.

or if we use derived form:

YOUNG_STUDENTS => A''(LT(I(AGE);20);(AGE,STUDENT))

Also we may define GOOD_STUDENTS as follows:

GOOD_STUDENTS => I(EQ(A(MARK);STUDENT)) x A(GT(I(MARK);14))

or if we use derived form:

GOOD_STUDENTS => A'' (GT(I(MARK);20);(MARK,STUDENT))

and YOUNG_BAD_STUDENTS as:

YOUNG_BAD_STUDENTS => YOUNG_STUDENTS - GOOD_STUDENTS

and YOUNG_GOOD_STUDENTS as:

YOUNG_GOOD_STUDENTS => YOUNG_STUDENTS x GOOD_STUDENTS

(2) name(arg1,arg2,...) => ff where argi are called variables and must appear in the functional form ff at least once. This definition has exactly the same role as the preceeding one except it introduces variables in functional forms. For example, let X be a variable,

ext(X) => P(I(X)) x I(P(X))

defines ext as a function which computes the tuples of values characterizing the entity node X. To evaluate this new function, we must instantiate the variable X. For example, ext(PERSON) or ext(STUDENT).

6- SOME GENERAL PROPERTIES ON THE SEMANTIC NETWORK

Before listing some of these properties, we first define how to express the cardinality constraint using MORSE language. The evaluation of each function at the point Xo, returns a set e of values. We will denote Card(e) the cardinal of this set. When Card(e) satisfies the equation $m<Card(e)<n$, we may write $Card(e) \in [m,n]$ or using an abusive notation $Card(e)=[m,n]$. The interval [m,n] is called the cardinalities of the variable set e and the values m,n are called the minimal and the maximal cardinalities respectively.

Using this new function Card, the Boolean functions, the primitives, the derived forms and the functional forms, we can state the following interesting properties which can be demonstrated easily.

G1 if $s(X,Y)=\Gamma$ then $I(Y) \subseteq I(X)$

G2 if $s(X,Y)=\Gamma$ and $s(X,Z)=\Gamma$ then $I(Y) + I(Z) \subseteq I(X)$

G3 $\forall X$, $I(S(X)) \subseteq I(X)$

Beware, $I(X) = I(S(X))$ does not imply $X = S(X)$

G4 if $s(X,Y)=\Gamma$ and $s(X,Z)=\Gamma$ and $Card(I(X)) + Card(I(Z)) > Card(I(X))$

then $I(Y) \times I(Z) \neq \underline{\phi}$

This latter result may be expressed using derived forms:

G5 $\forall x \in I(X)$, if $Card(S'(x;X)) > [1,1]$ then $I(Y) \times I(Z) \neq \underline{\phi}$

G6 if $G(X) = \{Y,Z\}$ then $I(X) = I(Y) \times I(Z)$

G4 and G5 formulas show us the correlation that exists between cardinalities and intersections. Using only the function Card, we can express the both constraints of cardinalities and intersections. Thus, checking that a person has only one name and several addresses is equivalent to check the following predicates:

$\forall x \in I(PERSON)$, $Card(P''(x;(PERSON,NAME))) = [1,1]$

$\forall y \in I(ADDRESS)$, $Card(P''(y;(PERSON,ADDRESS))) > [1,1]$

In the same way, the fact that PROF and INSTRUCTOR are exclusive can be

expressed by the following predicate:

$\forall x \in I(TEACHER)$, Card(S'(x;TEACHER)) = [1,1]

If a teacher may be a staff_member then,

$\exists x \in I(EMPLOYEE)$ / Card(S'(x;EMPLOYEE)) >= [1,1]

The case of STUDENT and INSTRUCTOR which are not directly linked to the same generic node, and discussed previously (section 2-3), could be solved by the following predicate:

$I(STUDEND) \times I(INSTRUCTOR) \neq \underline{\phi}$.

7- CONCLUDING REMARKS

In this paper, we have described two major topics. First, we have defined a Semantic Network with a few built-in predicates for which someone may add any predicate specific to his or her application. Nodes and arcs of this Semantic Network are placed into categories to make sure that each object is well defined. Some integrity constraints enhance the semantics of this model. As this model allows to represent both types and instances in the same schema, the classical distinction between schema and database desappears. To implement this kind of semantic network, it was suggested in [BOUZ83] to transform it in a relational database with a great variety of integrity constraints to capture the lost structural semantics of the first schema.

However, some simplifications are made and some problems are ignored. For example, in the Semantic Network, several different specializations are not allowed for a given node (i.e. we cannot see EMPLOYEE as TEACHER and STAFF_MEMBER on the one hand and as CONTRACTUAL_EMPLOYEE and TEMPORARY_EMPLOYEE on the other hand). Also recursive application of aggregation arcs are not studied and the category of constraints is not completely listed. The existential and universal quantifiers have to be more formalized.

Second, after reviewing functional programming, we have proposed a functional query language to manipulate this semantic network. It consists of primitives, derived forms and functional forms. This language permits both consulting, updating and expressing constraints. A very simple constructions permit to introduce variables and to define new functions. Specific properties of this language and general properties of the semantic network are deduced.

Finaly, like the famous telegraphic code of the same name, MORSE statements are not easily readable for non-experts in functional programming or in mathematical abstractions. Thus, to facilitate interactions, queries have to be automaticaly encoded and decoded just like morse messages are. Interfacing this symbolic language with the Natural Language would be an exciting subject certainly.

AKNOWLEDGEMENTS

I wish to thank Pr. Georges Gardarin and the other members of Sabre Project for the nice environment they have created for me to do this work. Also thanks to Beth for her tireless patience to always correct my horrible English along neverending weekends.

REFERENCES

SEMANTIC DATA MODELS

[BRAC77] BRACHMAN R.J. What's in a Concept: Structural Foundations for Semantic Networks. (Int. J. Man-Machine Studies (1977),9)

[GRIF82] GRIFFITH R. L. Three Principles of Representation for Semantic Networks (ACM TODS vol 7, no 3, sept 1982)

[HAML81] HAMMER M. & McLEOD D. Database Description with SDM: a Semantic Database Model. (ACM TODS vol 6, no 3, sept 1981)

[HEND75] HENDRIX G.G. Encoding Knowledge in Partitionned Networks. (in Associative Networks, Representation and Use of Knowledge by Computers, edit. by N V FINDLER Academic Press 1979)

[LEGE78] LEE R.M. & GERRITSEN R. Extended Semantic for Generalization Hierachies. (ACM SIGMOD 1978)

[MYLO81] MYLOPOULOS J. An overview of Knowledge Representation (SIGPLAN 16,1 1981)

[ROMY76] ROUSSOPOULOS N. & MYLOPOULOS J. Using Semantic Networks for Database Management ((Proceed. of VLDB Conf. Sept 1975)
[SCHU76] SCHUBERT L.K. Extending the Expressive Power of Semantic Networks (Artif. Intel. (1976, 7))
[SMSM77] SMITH J.M. & SMITH D.C.P. Databases Abstractions: Aggregation and Generalization (ACM TODS june 1977)
[WOOD75] WHOODS W.A. What's in a Link? Foundation for Semantic Networks. (in Representation and Understanding Studies in Cognitive Sces ed. by Bobrow and Collins (Academic Press 1975)

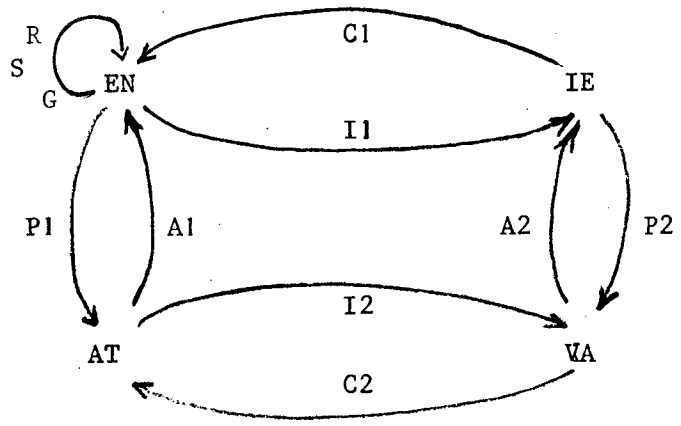
FUNCTIONAL PROGRAMMING

[BACK73] BACKUS J. Can Programming be liberated from the von Neumann Style? A Functional Style and its algebra of Programs (CACM vol 21, no 8, 1978)
[BACK81a] BACKUS J. The Algebra of Functional Programs: Function Level Reasoning Concepts (Proceed of colloq in Formalization of Programming Concepts, Peniscola, Spain 1981, ed Springer Verlag)
[BACK81b] BACKUS J. Function level Programs as Mathematical Objects (Proceed of the 1981 Conf. on Functional Programming languages and Computer Sces Oct 1981, Portsmouth, New Hampshire)
[BUFR79] BUNEMAN P. & FRANKEL R. FQL: A Functional Query Language (ACM SIGMOD Boston May 1979)
[BUNF81] BUNEMAN P. NIKHIL R. & FRANKEL R. A Practical Functional Programming System for Databases (Proceed of the 1981 Conf. on Functional Programming Languages and Computer Sces Oct 1981, Portsmouth, New Hampshire)
[BUNF82] BUNEMAN P. FRANKEL R. & NIKHIL R. An Implementation Technique for Database Query Languages (ACM TODS vol 7, no 2 , june 1982)
[SCHI81] SCHIPMAN D.W. The Functional Data Model and the Data Language DAPLEX (ACM TODS vol6, no 1, March 1981)

GENERAL TOPICS

[BOUZ83] BOUZEGHOUB M. & GARDARIN G. The Design of an Expert System for Data Bases Design (in Proceed. of Special Workshop on New Applications of Data Bases, Cambridge (UK) Gardarin and Gelenbe edit. Academic Press 1983)
[CODD70] CODD E.F. A Relational Model for Large Shared Data Banks (CACM vol 6, no 13)
[CLME81] CLOCKSIN W.F. & MELLISH C.S. Programming in Prolog (Springer-Verlag edit. 1981)
[WIHO81] WINSTON P.H. & HORN B.K.P. LISP (Addison-Wesley Publ. Co. 1981)

APPENDIX



The following table shows the permissible products of functions. For the other compositions, the rule is very simple: the results of the two functions must belong to the same node category.

* →	A_1	P_1	A_2	P_2	C_1	I_1	C_2	I_2	G	S	R
A_1		1					1				
P_1	1				1				1	1	1
A_2				1				1			
P_2			1			1					
C_1			1			1					
I_1	1				1				1	1	1
C_2				1				1			
I_2		1					1				
G	1				1				1	1	1
S	1				1				1	1	1
R	1				1				1	1	1